Binary Semaphores

Semaphores are used to signal/synchronize tasks as well as protect resources.

A binary semaphore can (and should) be used as a means of signaling a task. This signal can come from an interrupt service routine or from another task. A semaphore is an RTOS primitive and is guaranteed to be thread-safe.

Design Pattern

Wake Up On Semaphore

The idea here is to have a task that is waiting on a semaphore and when it is given by an ISR or an other task, this task unblocks, and runs its code. This results in a task that usually sleeping/blocked and not utilizing CPU time unless its been called upon. In FreeRTOS, there is a similar facility provided which is called 'deferred interrupt processing'. This could be used to signal an emergency shutdown procedure when a button is triggered, or to trigger a procedure when the state of the system reaches a fault condition. Sample code below:

```
/* Declare the instance of the semaphore but not that you have to still 'create' it which i
SemaphoreHandle_t xSemaphore;

void wait_on_semaphore_task(void * pvParameters) {
    while(1) {
        /* Wait forever until a the semaphore is sent/given */
        if(xSemaphoreTake(xSemaphore, portMAX_DELAY)) {
            printf("Semaphore taken\n");
            /* Do more stuff below ... */
        }
    }
}

void semaphore_supplier_task(void * pvParameters) {
    while(1) {
        if(checkButtonStatus()) {
            xSemaphoreGive(xSemaphore);
        }
        /* Do more stuff ... */
    }
}

int main()
{
    /* Semaphore starts 'empty' when you create it */
    xSemaphore = xSemaphoreCreateBinary();

    /* Create the tasks */
    const uint32_t STACK_SIZE_WORDS = 128;
    xTaskCreate(wait_on_semaphore_task, "Waiter", 512, NULL, PRIORITY_LOW, NULL);
    xTaskCreate(semaphore_supplier_task, "Supplier", 512, NULL, PRIORITY_LOW, NULL);
```

```
32
33      /* Start Scheduler */
34      vTaskStartScheduler();
35  }
```

Code Block 1. How to use Semaphores and use as a wake up pattern

Semaphore as a flag

The idea of this is to have a code loop that checks the semaphore periodically with the 'block time' of your choice. The task will only react when it notices that the semaphore flag has been given. When your task takes it, it will run an if statement block and continue its loop. Keep in mind this will consume your flag, so the consumer will loop back and check for the presence of the new flag in the following loop.

```
1  void vWaitOnSemaphore( void * pvParameters )
2  {
3      while(1) {
4          /* Check the semaphore if it was set */
5          if(xSemaphoreTake(xSemaphore, 0)) {
6              printf("Got the Semaphore, consumed the flag indicator.");
7              /* Do stuff upon taking the semaphore successfully ... */
8          }
9
10         /* Do more stuff ... */
11     }
12 }
```

Code Block 2. Semaphores as a consumable flag

Interrupt Signal from ISR

This is useful, because ISRs should be as short as possible as they interrupt the software or your RTOS tasks. In this case, the ISR can defer the work to a task, which means that the ISR runtime is short. This is important because when you enter an interrupt function, the interrupts are disabled during the ISRs execution. The priority of the task can be configured based on the importance of the task reacting to the semaphore.

> You may not want to defer interrupt processing if the ISR is so critical that the time it takes to allow RTOS to run is too much. For example, a power failure interrupt.

```
1  void systemInterrupt() {
2      xSemaphoreGiveFromISR(xSemaphore);
3  }
4
5  void vSystemInterruptTask(void * pvParameter) {
6      while(1) {
7          if(xSemaphoreTake(xSemaphore, portMAX_DELAY)) {
8              // Process the interrupt
9          }
```

```
    }
}
```

Code Block 3. Semaphore used within an ISR

> ***NOTICE:*** *The* ***FromISR*** *after the* ***xSemaphoreGive*** *API call? If you are making an RTOS API call from an ISR, you must use the* ***FromISR*** *variant of the API call. Undefined behavior otherwise like freezing the system.*